RELATORIO
PROGRAMAÇÃO PARALELA

PROFESSOR: DANIEL WEINGAERTNER
ALUNOS:
GUSTAVO TOSHI KOMURA GRR20102342
LUIS ALEXANDRE DESCHAMPS BRANDÃO GRR20085461

Solução

A estrutura utilizada consiste de 2 double que são:
- fxy: valor do resultado da função f(x,y).
- “valor”: valor do resultado de cada iteração calculados a partir da função de atualização.

O valor de fxy é determinado pela função f(x,y) = 4π^2sin(2πx)sinh(2πy).
O valor de “valor” é determinado pela função
u(x,y) = 1/((2/hx^2)+(2/hy^2)+k^2)(f(x,y) + (1/hx^2)(u[x-1,y] + u[x+1,y]) + (1/hy^2)(u[x,y-1] + u[x,y+1])).

A malha (plano cartesiano) foi organizada em uma matriz de estruturas que ao ser inicializada são calculados os valores de fxy baseando-se nos valores de entrada nx e ny, e o “valor” de cada ponto é inicializado com “0” por causa da solução inicial.
Ao se executar o programa é verificado se todos os parâmetros foram passados e se eles estão corretos, depois é verificado qual o método de solução escolhido (Jacobi ou Red-Black Gaussian)
Para o método Red-Black Gaussian, inicialmente é armazenado o tempo atual para depois se fazer o cálculo do tempo de execução, e em seguida é criada uma matriz (nx + 1)x(ny + 1) e são executados n iterações, que é o valor passadas por parâmetro, onde em cada iteração existem dois loops paralelos, o primeiro calcula a função de atualização nos pontos red e o segundo nos pontos black. Por fim depois de passar por todas as iterações é pego o tempo atual e depois é feito o cálculo do tempo de execução, e no final é calculado o resíduo e a solução é gravada em um arquivo.

Implementação inicial e melhoramentos

Na implementação inicial era utilizada uma estrutura de 4 doubles que eram x, y, fxy e e valor, onde x e y são as coordenadas no plano cartesiano. Esta implementação conseguia escalonar razoavelmente bem, no entanto o tempo de sua execução era grande, provavelmente porque a quantidade de dados passados da memória para a cache era muito grande, provocando várias caches miss durante a execução do programa e também pelo alto trafego de dados no barramento.

Para se chegar a versão final fizemos a remoção de “x” e “y” da estrutura, com isso o tempo de execução caiu pela metade e também fizemos um melhoramento no algoritmo Red-Black Gaussian, onde substituímos nos loops paralelos, “ifs” que verificavam se “i” era para ou impar, por “mods” e com isso o tempo de execução chegou a diminuir mais ou menos 1 segundo. Estes “ifs” serviam para verificar se estávamos em uma linha que o red era o primeiro ou o segundo da linha.

Tentativas de melhoramento

1 - Trocar o uso de uma matriz por um vetor, no entanto o tempo de execução foi maior, ou seja o desempenho caiu.

2 - Ao invés de armazenar o valor de fxy fazer o calculo a cada iteração, no entanto o tempo de execução foi na maior parte das vezes mais que o dobro do tempo. Entretanto, nesse caso, o problema

escalava de maneira adequada; quando dobrava-se o número de threads o tempo de processamento era dividido por dois. Isso mostra que o nosso problema é memory bound.

3 - Implementar Red-Black Gaussian sobre dois vetores, um vetor representado os pontos red e o outro os pontos black, desta forma um vetor seria somente para leitura e outro somente para escrita. O problema desta implementação tem relação com a tentativa número 1, ou seja, trocar vetor por matriz deixa o programa mais lento e também por causa da dependência de dados, pois por exemplo suponha que estamos calculando um ponto red e para calcular este ponto na função de atualização precisamos do valor de fxy e este valor é armazenado na estrutura, no entanto esta estrutura está no vetor red, lugar onde deveríamos somente escrever e não ler. Deste modo para evitar este problema seria possível fazer o cálculo de fxy a cada iteração apesar do problema da tentativa 2.

Resultados:

Todos os testes foram executados na latrappe que possui um Intel(R) Xeon(R) CPU E5-2680 v2 @ (20 cores 2.80GHz), onde a versão inicial foi executada em 30 rodadas de testes e a versão inicial melhorada foi executada em 10 rodadas de testes.

Resultados que não forma colocados neste relatório pelo propósito de que eles não deixavam a solução escalonável, pudemos perceber que quando as matrizes possuíam um tamanho pequeno ao se aumentar o número de threads o desempenho caia e isso se deve ao fato de que com o aumento de processadores, a comunicação aumenta e como a quantidade de dados é pequena, o tempo gasto com comunicação se torna mais relevante para o desempenho do que o tempo de processamento, ou seja, se gasta mais tempo comunicado do que processando.

Fazendo um comparativo entre a *Tabela 1 - Resultado Testes Versão Inicial* que representa a versão inicial com 30 rodadas de testes e a *Tabela 2 - Resultado Testes Versão Melhorada* que representa a versão inicial melhorada, podemos perceber que com a versão inicial melhorada chegamos a diminuir mais ou menos 6 vezes o tempo de execução com relação ao tempo da versão inicial.

Tabela 1 - Resultado Testes Versão Inicial

Processadores / Tamanho	1024x1024	1025x1025	2048x2049	2049x2049
1	31.178097	45.422426	122.802597	171.380766
2	14.978149	23.897554	59.220685	102.041294
4	8.532282	12.80174	32.263015	45.09318
8	8.982852	11.228148	21.964727	25.397213

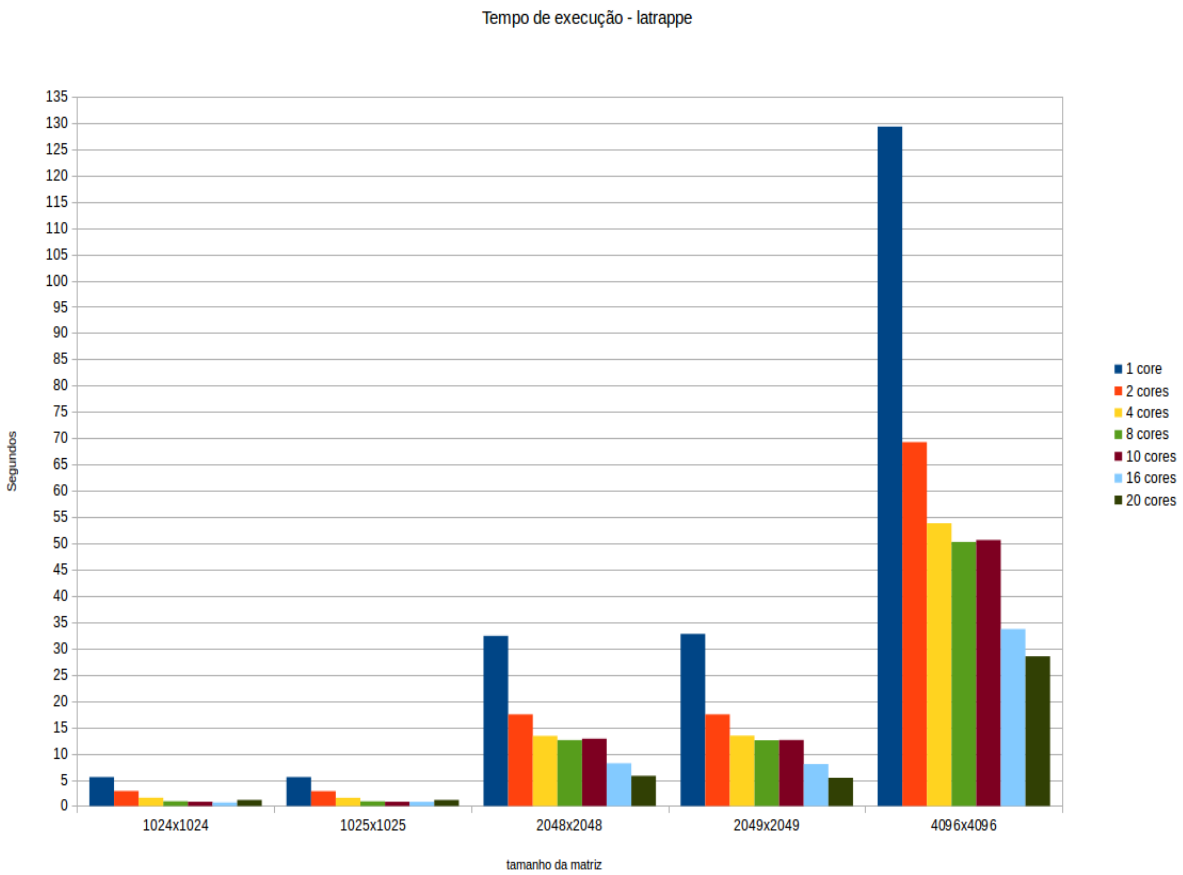
Tabela 2 - Resultado Testes Versão Melhorada

Processadores / Tamanho	1024x1024	1025x1025	2048x2049	2049x2049
1	5.491082	5.487897	32.38068	32.750545
2	2.854395	2.828683	17.42949	17.432038
4	1.563692	1.549599	13.343931	13.394752
8	0.894996	0.885567	12.54546	12.512198

Fazendo um comparativo entre os vários tamanhos de matrizes com várias quantidade de threads na versão inicial melhorada no *Gráfico 1 - Tempo de Execução*, podemos perceber que não é

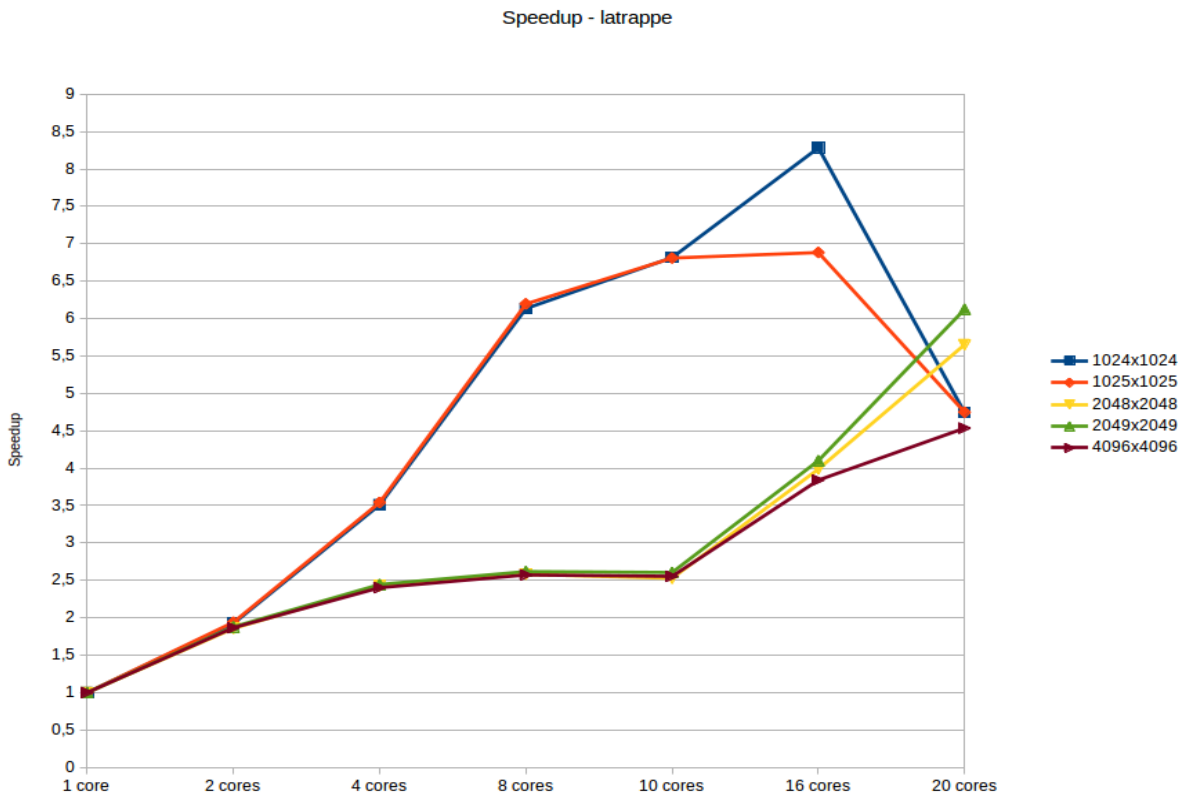
compensatório escalonar com mais de 8 threads para matrizes de tamanho 1024 e 1025, pois os tempos de execução com o aumento de threads ficam próximos ou até piores dos que tem menos threads. Também se pode perceber no gráfico que a versão inicial melhorada é razoavelmente escalonável, mas pelo problema ser memory bound, ao usarmos 10 threads, estamos na situação onde o barramento está saturado e vemos uma perda de desempenho.

Gráfico 1 - Tempo Execução



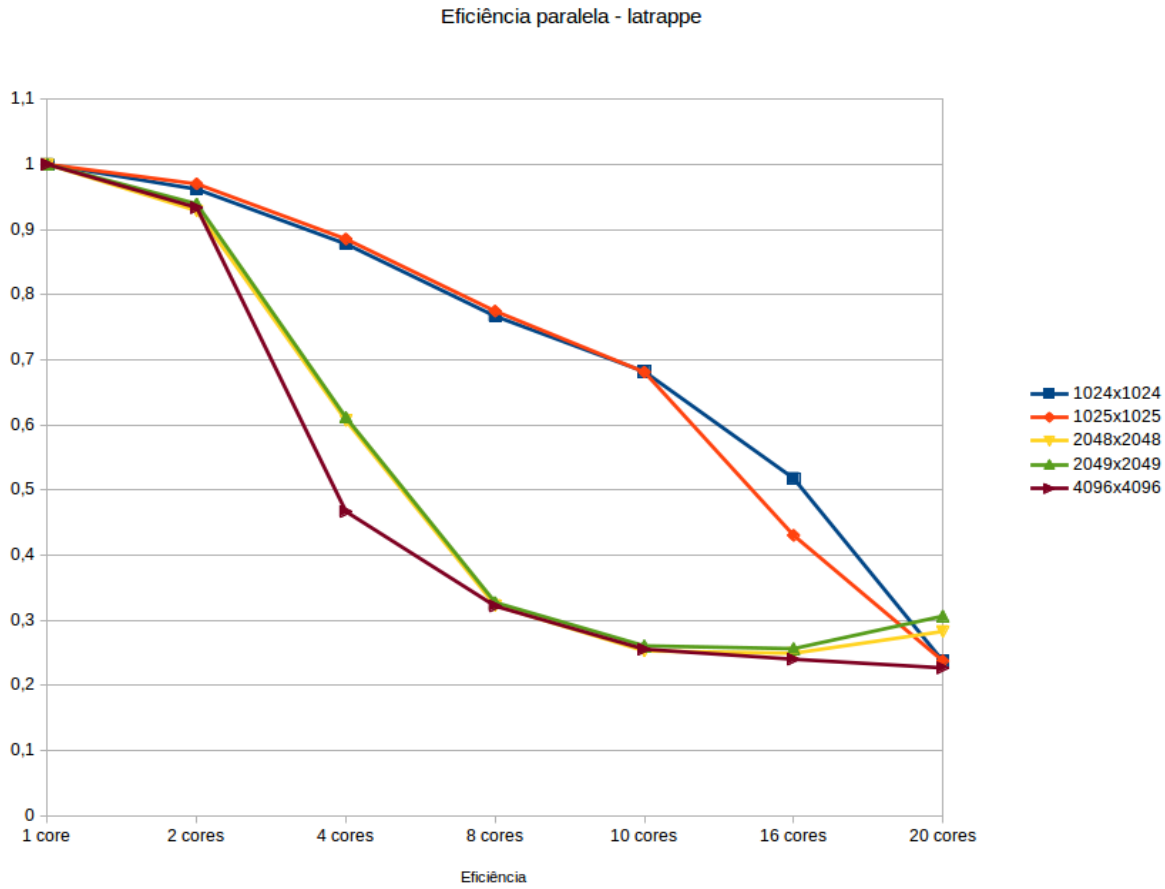
Fazendo um comparativo entre os vários tamanhos de matrizes com várias quantidade de threads na versão inicial melhorada no *Gráfico 2 - Speedup*, podemos perceber que até 2 threads o speedup de todos os tamanhos de matrizes foram parecidos, sendo que após 2 threads as matrizes de tamanho 1024 e 1025 aumentaram o seu speedup mais do que os outros, no entanto ao chegarem em 16 threads o seus speedups caíram drasticamente chegando a ficar somente acima da matriz de tamanho 4096.

Gráfico 2 - Speedup



Fazendo um comparativo entre os vários tamanhos de matrizes com várias quantidade de threads na versão inicial melhorada no *Gráfico 3 - Eficiência Paralela*, podemos perceber que até 2 threads a eficiência se manteve quase constante para qualquer tamanho de matriz, no entanto acima de 2 threads a eficiência para matrizes de tamanho 2048, 2049 e 4096 pioraram drasticamente se recuperando quando passaram por 8 threads. Também podemos perceber que ao passar por 16 threads as matrizes de tamanho 2048 e 2049 aumentaram a sua eficiência.

Gráfico 5 - Eficiência Paralela



Trabalhos futuros

Para trabalhos futuros poderíamos ganhar desempenho se quebrarmos a estrutura em duas, sendo uma estrutura armazenamento o valores de fxy e outra que possua somente o “valor”. Com esta implementação poderia haver um ganho, pois fxy seria somente para leitura após serem inicializados e desta forma poderíamos implementar a tentativa 3 sem nenhuma dependência de dados. Voltando ao exemplo de calcularmos a função de atualização em um ponto red, não teríamos nenhuma dependência de dados, pois os valores fxy somente seria lido da nova estrutura e os valores do stencil também somente seriam lidos das estruturas que estão no vetor black e desta forma só iriamos escrever na estrutura red.

Conclusão

Podemos perceber pelos resultados que com o aumento no número de threads a implementação paralela melhora a velocidade de execução para matrizes de tamanhos grandes e isso se deve ao fato de que o processamento dos dados é bem distribuído entre as threads, no entanto para matrizes de tamanho pequeno o melhoramento do desempenho não foi alcançado, pois o tempo de comunicação tornou-se mais relevante que o tempo de processamento.